CSE 351: The Hardware/Software Interface

Section 1

Intro, C programming, C tools

Introduction

* My name is Matt Dorsett
* It's my third year in the computer engineering program
* Have done software development for Pariveda Solutions and Expedia
* Will be an intern at Qualcomm next summer in San Diego
* Interests include embedded systems, robotics, control theory, operating systems
* I have office hours on Monday from 3:30-4:30 in CSE 002, but I will always respond to emails and can meet with you by appointment
* Contact: discussion board or by email (madman2@cs)

Course Tools

- Use whatever works best for you: the CSE home VM, attu, the instructional Linux machines, or your own Linux installation (we won't provide support if you go this route, though)
- From pretty much any machine, you can use PuTTY (Windows) or an SSH client (OS X, Linux, iOS, Android, etc.) to access attu
  - Via SSH: ssh [username]@attu.cs.washington.edu

Course Tools

- We'll be using the GNU C Compiler (gcc) for compiling C code in this course, which is available on pretty much every platform except Windows (unless through Cygwin)
- For an editor, use whatever makes you comfortable; Emacs, Vim, gedit, and Eclipse are good choices

Unix Commands

- We're going to assume that you know some basic Unix commands; there are many guides online if you need additional help such as this one
- cd: change directory
  - Example: `cd path/to/directory`
- pwd: print working directory
  - Example: From my home directory on attu, `pwd` prints out `/homes/iws/snowden`
- ls: list directory contents
  - Example: `ls ..` (list the directory one above this one)
- chmod: change mode (permissions)
  - Example: `chmod +x file` (make file executable)

Compiling C Code

- There are two steps to get from a C source file to an executable file: compiling and linking
- To compile a source file with GCC, use the `-c` option: `gcc -c example.c`
  - This will produce a corresponding example.o file, which contains the machine code for the example.c source file
- To link object files into an executable with GCC, list them as arguments: `gcc -o example example.o […]`
  - Here the -o option specifies what to name the output; it will be an executable file called “example”

Compiling C Code

- It's also possible to combine the two steps: `gcc -o example example.c`
  - This will accomplish both the compilation and the linking at once
  - Why might it be a good idea to separate these two steps?
- GCC takes a number of flags, which you will see/have seen with lab 0
  - `-g` to include debugging symbols
  - `-Wall` to warn about all recognized problems
  - `-std=gnu99` to use the C99 standard instead of the C89 standard, which is just a couple years out of date
  - Example: `gcc -g -Wall -std=gnu99 -o example example.c`

A Basic C Program

- The Hello World of C:

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("Hello World\n");
  return 0;
}
```

A Basic C Program

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("Hello World\n");
  return 0;
}
```

- The first line is a *header* inclusion
- Headers provide declarations (but not normally definitions) of other code
  - stdio.h contains the declaration of the `printf` function, which is used for printing to the console

A Basic C Program

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("Hello World\n");
  return 0;
}
```

- On Linux, you can look under /usr/include to see the contents of these header files
- To refer to headers that aren't part of "special" directories, put the path to them in quotes
  - As an example, `#include "path/to/header.h"`

A Basic C Program

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("Hello World\n");
  return 0;
}
```

* The next part of the file is the declaration of the entry point for the program: `main()`
  * `main()` takes two parameters, the first of which is the number of strings contained in the second parameter. `argv` is an array of the arguments to the program

A Basic C Program

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("Hello World\n");
  return 0;
}
```

* The `printf()` function prints to the console. It is equivalent to Java's `System.out.printf()` and requires that you insert a newline explicitly

A Basic C Program

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("Hello World\n");
  return 0;
}
```

- Finally, `return 0` indicates the status code of the program when it exits
- A status code of 0 indicates success, whereas other numbers have a different meaning
  - `errno.h` includes the names of many status codes, which are documented in "`man errno`"

A Basic C Program

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("Hello World\n");
  return 0;
}
```

* Let's compile and run the program

Formatting Output

- In C, there is no easy way to concatenate strings as there is in Java. Instead, `printf()` supports a number of format codes
- Example: `int val = 10; printf("%d\n", val);`
  - %d is the format code for ints, so the above code will print “10” with a newline
- Other format codes: %f for floats and doubles, %s for strings, %x for hexadecimal values, %p for pointers. See the cplusplus site for more info

Formatting Output

- A few different scenarios:

```
printf("There are %d students enrolled "
       "in the class\n", 88);
printf("The course number for this "
       "class is %s\n", "CSE 351");
printf("If you want a %f in %s, you'll "
       "need to work for it\n", 4.0,
       "CSE 351");
```

Man Pages

- Much of the functionality of Linux is documented in *man pages*. Man pages are manuals describing how a variety of commands, functions, and so forth work
- As an example, take a look at `man ssh`. This describes how the `ssh` command works
- For C functions, look in section 3; that is, use `man 3 [topic]`, so `man 3 printf` for the `printf()` function

Debugging

- The best way of debugging C programs is to use GDB (not printf statements!)
- GDB is the GNU debugger, and it does a variety of amazing things. To use it, compile your program using the -g option (to include debugging symbols) and then run in under GDB with `gdb ./example`
- Let's run the hello world program from before under GDB

Debugging

- Use the “p” (print) command within GDB to print out values of variables and their addresses
- Use the “b” (breakpoint) command to set a breakpoint at a particular line/file/function, e.g. “b 79” to break execution at line 79 in the current file
- Use the “c” (continue) command to resume execution after hitting a breakpoint
- Use the “d” (delete breakpoint) command to remove breakpoints, e.g. “d 1” to delete breakpoint 1
- Use the “list” command to output the code with line numbers in the current file. “list [line-#]” will list code from the given line; press Enter to see more code

Debugging

- Use the “x” (examine) command within GDB to examine memory at a certain address (more useful in later labs)
- Use the “r” (run) command to execute the program
- Use the “s” (step) command within GDB to execute one C statement
- Use the “n” (next) command to execute one C statement, skipping over function calls

Debugging

- Use the “bt” (backtrace) command within GDB to print out the current call stack
- Use the “frame” command jump to the indicated stack frame, e.g. “frame 3” for stack frame 3. Use this in combination with the “bt” command
- When setting breakpoints, you can specify a condition so that the debugger only breaks if the condition is met, e.g. “b example.c:83 if x == 10” will set a breakpoint at line 83 of example.c that will activate only when x is 10

Your Turn

- Working in pairs/groups, download the two .c files for this section from the course calendar and use GDB to debug and fix the problems using the techniques given in the source files
  - Work first on conditional.c, then on backtrace.c
  - Alternatively, if you haven't completed lab 0, now would be a good time to do it
- Be sure to ask for help if needed!
- GDB Cheat Sheet:
  - http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf